策略模式

> 定义一系列算法, 把他们一个个封装起来, 并使它们可以相互替换

目的

- 将算法的使用和算法的实现分离开来
- 可以有效的避免多重条件选择语句
- 更有效的复用代码

接下来我们来实现一个需求, 指定一周计划, 并根据当天是周几来决定该干什么

```
function doWhat(weekDay) {
    if(weekDay == 1) {
        console.log('看书')
    }else if(weekDay == 2) {
        console.log('爬山')
    }else if(weekDay == 3) {
        console.log('看电影')
    }else if(weekDay == 4) {
        console.log('玩游戏')
    }else if(weekDay == 5) {
        console.log('写博客')
    }else if(weekDay == 6) {
        console.log('休息')
    }else {
        console.log('休息')
    }
}

dowhat(2) // 爬山
```

以上方式虽然可以实现需求, 但是, 如果我要加入其它需求就不得不修改weekDay函数, 而且条件分支越多, 代码阅读起来就越吃力

接下来我们换种方式

```
let dosome = {
    1: '看书',
    2: '爬山',
    3: '看电影',
    4: '玩游戏',
    5: '写博客',
    6: '休息',
    7: '休息'
}

function dowhat(weekDay){
    console.log(dosome[weekDay])
}
```

```
dowhat(1) // "看书"
```

现在看起来是不是清爽多了, doWhat 里面封装的是算法的实现, 是抽离出来的算法实现时的共性, 而 dosome 则是封装了一系列算法的对象, 每个算法都不一样, 只要后面有其他算法加入, 我们也只要在 dosome里面添加或修改, 并不需要去了解doWhat里面的实现逻辑

再举个栗子

简单实现表单验证邮箱, 手机号

```
let strategy = {
    isEmail(value){
        const re = /^(\w-*\.*)+@(\w-?)+(\.\w{2,})+$/
        return re.test(value)
    },
    isPhone(value){
        const re = /^[1][3,4,5,7,8][0-9]{9}$/
        return re.test(value)
    }
}

// 实现
strategy.isEmail('17377774547') // false
strategy.isPhone('17377774547') // true
```

如果, 其他页面也还有验证操作, 直接将这个对象引入就可以直接使用, 并不需要重新写验证函数